May 30, 2008

RT0796
Computer Science 9 pages

Research Report

Finding Bugs in Java Native Interface Programs

Goh Kondoh, Tamiya Onodera

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Finding Bugs in Java Native Interface Programs

Goh Kondoh Tamiya Onodera
Tokyo Research Laboratory
IBM Research
1623-14, Shimotsuruma, Yamato-shi
Kanagawa-ken, Japan
+81-46-215-4584, +81-46-215-4645
{gkondo,tonodera}@jp.ibm.com

ABSTRACT

In this paper, we describe static analysis techniques for finding bugs in programs using the Java Native Interface (JNI). The JNI is both tedious and error-prone because there are many JNI-specific mistakes that are not caught by a native compiler. This paper is focused on four kinds of common mistakes. First, explicit statements to handle a possible exception need to be inserted after a statement calling a Java method. However, such statements tend to be forgotten. We present a typestate analysis to detect this exception-handling mistake. Second, while the native code can allocate resources in a Java VM, those resources must be manually released, unlike Java. Mistakes in resource management cause leaks and other errors. To detect Java resource errors, we used the typestate analysis also used for detecting general memory errors. Third, if a reference to a Java resource lives across multiple native method invocations, it should be converted into a global reference. However, programmers sometimes forget this rule and, for example, store a local reference in a global variable for later uses. We provide a syntax checker that detects this bad coding practice. Fourth, no JNI function should be called in a critical region. If called there, the current thread might block and cause a deadlock. Misinterpreting the end of the critical region, programmers occasionally break this rule. We present a simple typestate analysis to detect an improper JNI function call in a critical region.

We have implemented our analysis techniques in a bug-finding tool called BEAM, and executed it on opensource software including JNI code. In the experiment, our analysis techniques found 86 JNI-specific bugs without any overhead and increased the total number of bug reports by 76%.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *correctness proofs, reliability, validation*. D.3.3 [**Programming Languages**]: Language Constructs and Features – *constraints, data types and structures, polymorphism, procedures, functions, and subroutines*.

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

General Terms

Algorithms, Performance, Design, Reliability, Experimentation, Languages, Verification.

Keywords

Java Native Interface, static analysis, typestate analysis

1. INTRODUCTION

A foreign function interface (FFI) allows code written in one language to call code written in another language. Many programming languages support their own FFIs, and Java's version is called the Java Native Interface (JNI). However, the JNI is tedious to use and error-prone. For example, the programmer's guide [10] devotes an entire chapter, *Traps and Pitfalls*, to 15 of the most common programming errors (Table 1). To express a simple expression that can be written as just a few terms in Java, JNI typically requires several lines in native code. Also, some JNI functions must be called in a specific order. As a result, code using the JNI is more likely to have bugs than code without JNI calls. These JNI mistakes are not caught by the compiler. We encountered these problems with JNI programming while we were developing a static analysis tool for Java in BEAM [3]. In an early phase of the development, we integrated a Java parser written in Java with the BEAM modules written in C and C++. During this development we encountered most of the problems described in Table 1, and that motivated us to create a JNI bug-finding tool.

Table 1 spans from high-level design issues to bad coding practices at low levels. The design issues 5, 6, 7, 14 and 15 are beyond our scope because they depend on the target software and cannot be checked automatically by a static analysis tool. We worked on the other problems, especially those not covered by prior research, and prioritized them according to their severity. We selected the problems marked with symbol 'X' and also added the problem of calling a JNI function from a critical region, which is described later.

A type inference system [6] is a useful approach for Problems 2 and 3. We implemented such a system and found it somewhat useful during the coding phases. However, it did not perform well in our experiments (for reasons explained in Section 6.2). Problem 4 can easily be detected by a syntax checker or a compiler warning (such as a –Wconversion warning). BEAM already included a checking tool for this kind of bug, but it also failed to produce some of the relevant warning messages in our experiments. Therefore, we do not further discuss these tools in this paper.

Problems 8, 9, 10, and 12 are future work, mostly due to limited interest from our customers and limited need based on our experiences. Each of these problems would also require a customized syntax checker, in contrast to allowing a generalized approach based on a standard dataflow analysis framework.

Thus, our goal was to find the errors related to Problems 1, 11, and 13 and the problem of calling a JNI function in a critical region. This paper's contributions are:

- We present static analysis techniques for JNI programs to detect:
  - mistakes of error checking,
  - memory leaks,
  - invalid uses of a local reference, and
  - JNI function calls in critical regions.
- We describe the implementation details for these techniques and experimental results with some benchmark programs.

Table 1: Traps, pitfalls, and the coverage of our tools and existing work. The second and third columns correspond to our tool and prior work.

Problems	Our focus	Prior work
1. Error Checking	X	
2. Passing Invalid Arguments to JNI Functions		[6] [4]
3. Confusing jclass with jobject		[6]
4. Truncating jboolean Arguments		compiler
5. Boundaries between Java Application and Native Code		
6. Confusing IDs with References		
7. Caching Field and Method IDs		
8. Terminating Unicode Strings		
9. Violating Access Control Rules		
10. Disregarding Internationalization		
11. Retaining Virtual Machine Resources	X	
12. Excessive Local Reference Creation		
13. Using Invalid Local References	X	
14. Using the JNIEnv across Threads		
15. Mismatched Thread Models		

This paper is organized as follows: In Section 2, we describe our analysis target JNI and define the kinds of mistakes our analysis should detect. In Section 3, we present our analysis techniques for detecting these problems. In Section 4, we give implementation details about our analytic tool and experimental results on some benchmarks. In Section 6 we compare our system to other research and we conclude with the utility of our analysis in Section 7.

2. JNI MISTAKES

In this section, we describe and give examples of the four common mistakes we try to detect. We selected these four because they seem common in open source software and because we had experiences of encountering them.

2.1 Error Checking

In Java, a method declaration explicitly includes one or more types of exceptions which may be thrown. A method invocation expression can appear in a try block that is followed by catch clauses corresponding to the exception types it can throw. These correspondences among the types of thrown exceptions, the types of caught exceptions, and the exceptions in the method declarations are checked by a compiler for Java programs.

Using JNI, a programmer can write native code calling a Java method that can throw an exception. Such a thrown exception is not related to the exceptions or try-catch statements in C++. Rather, it comes from the JNI function `ExceptionOccurred`. Therefore, JNI programmers need to insert exception checking code which works in the same way as Java's catch statement after a Java method invocation. This exception checking code is often forgotten because no compiler will notice that it is missing.

In order to invoke a Java method from native code, the following three steps are required for an instance method:

- The native code first calls `GetMethodID`, which performs a lookup for the method in the given class. The lookup is based on the name and type descriptor of the method. If the method does not exist, `GetMethodID` returns `NULL` and a `NoSuchMethodError` instance is created in a pending state.
- If the method was found in the previous step, the native code then calls `Call<`*`Type`*`>Method` where *<Type>* is the return type of the method. The receiver object, the method ID and the actual arguments are passed to this JNI function.
- The native code checks if an exception was raised in the previous step by calling the JNI function `ExceptionCheck` or `ExceptionOccurred`. If an exception was thrown, the native code clears the exception after handling the error or returns to Java with the exception pending.

These steps are very similar to those of reflected method invocations in Java except for the third one. For a reflection call in Java, a try-catch statement must be placed to handle the thrown exception of type `java.lang.reflect.InvocationTargetException`. In native code, the exception of a Java type cannot be caught by any language construct such as the try-catch statement in C++.

Figure 1 illustrates how to invoke the Java methods `foo()` and `bar()` on an object reference `obj` in C++ and how to check for the occurrence of an exception. Looking only at this native code, neither programmers nor compilers can tell whether or not the method `foo()` can throw an exception. However, this code never ignores the thrown exception because it conservatively assumes that `foo()` can throw an exception. Not all methods throw an

exception, but most of them can. According to [10], programmers are required to perform an explicit exception check after every JNI function call that could possibly throw an exception. This example is safe because the exception thrown from `foo()` will be never ignored.

```
jclass cls = env->GetObjectClass(obj);
jmethodID mid =
env->GetMethodID(cls, "foo", "()V");
env->CallObjectMethod(obj, mid);
if (env->ExceptionCheck()) {
  /* error handling */
  env->ExceptionClear(); /* or return */
}
mid = env->GetMethodID(cls, "bar", "()V");
env->CallObjectMethod(obj, mid);
```

Figure 1: A native code example calling a Java method

Figure 2 shows an example programmers tend to write. This example does not include the exception checking between the two Java method invocations. Note that the second invocation ignores an exception if it is raised by the first one. Java virtual machine features dynamic checks for JNI function calls if the -Xcheck:jni option is enabled. IBM J9 VM [2] with this option generates warnings for this kind of code even if it is executed without throwing an exception. If the programmer of this code is sure that `foo()` never throws an exception, omitting the exception checking may make sense. However, the checking is required in general.

```
jclass cls = env->GetObjectClass(obj);
jmethodID mid =
env->GetMethodID(cls, "foo", "()V");
env->CallObjectMethod(obj, mid);
mid = env->GetMethodID(cls, "bar", "()V");
env->CallObjectMethod(obj, mid);
```

Figure 2: An example possibly ignoring an exception

Furthermore, JNI functions can be called by a helper function as well as a top-level native method implementation function in native code. For example, in Figure 3, just looking at the main function, we do not know whether or not an exception handling mistake exists after returning from the helper function `CallFooWrapper`. Although `CallFooWrapper` does not check if an exception is thrown, it does not always mean an error because the exception can be correctly returned to Java if `CallFooWrapper` is called at the end of the native method execution. Therefore, in order to detect errors at helper function call sites we need to have an interprocedural analysis. Fortunately, this interprocedural analysis would be easy because it would not require context sensitivity: Whatever argument is passed to a function, we can assume that a JNI function to be called inside can produce an exception. This is analogous to that a Java method which declares an exception must be always called in an appropriate try-catch statement or a method rethrowing it.

```
void CallFooWrapper(JNIEnv *env, jobject obj)
{
  jclass cls = env->GetObjectClass(obj);
  jmethodID mid =
  env->GetMethodID(cls, "foo", "()V");
  env->CallObjectMethod(obj, mid);
}
void main(JNIEnv *env)
{
  jobject obj1 = …;
  jobject obj2 = …;
  …
  CallFooWrapper(env, obj1);
  /* should exception checking be here?*/
  jclass cls = env->GetObjectClass(obj2);
  jmethodID mid =
  env->GetMethodID(cls, "foo", "()V");
  env->CallObjectMethod(obj2, mid);
}
```

Figure 3: An example calling a function possibly throwing an exception

2.2 Retaining Virtual Machine Resource

The second type of common mistakes is retaining virtual machine resource. Native code can dynamically allocate a Java virtual machine resource by the allocator functions:

- `GetStringChars(JNIEnv *env, jstring str,jboolean *isCopy)`
- `GetStringUTFChars(JNIEnv *env, jstring str,jboolean *isCopy)`
- `Get<Type>ArrayElements(JNIEnv *env, <ArrayType> array,jboolean *isCopy)` where `<Type>` is one of Boolean, Byte, Char, Short, Int, Long, Float, and Double

These functions return NULL if and only if invocation of them has thrown an exception, whose type is `OutOfMemoryError`.

The resource must be freed later by the corresponding deallocator functions:

- `ReleaseStringChars`
- `ReleaseStringUTFChars`
- `Release<Type>ArrayElements`

Programmers tend to forget calling deallocator functions, typically in an exception handling path.

In addition, some programmers have a misunderstanding about the third parameter `isCopy` of the allocator JNI functions. The `isCopy` parameter is of the pointer-to-jboolean type. During an allocator call, `JNI_TRUE` is assigned to `*isCopy` if a copy has been made for the return value. Otherwise, `JNI_FALSE` is assigned. Some programmers think that if `*isCopy` is `JNI_FALSE` (e.g. Figure 4) or if they pass NULL to `isCopy` (e.g. `GetStringChars(env, jstr, NULL)`) the returned resource does not need to be released later. However, regardless

of `isCopy` or `*isCopy`, the corresponding deallocator function should be called.

Of course, virtual machine resources should not be released more than once, or accessed after freed. We deal with these invalid uses of deallocated resources as well as leaks.

```
jboolean isCopy;
const char *cstr =
    (*env)->GetStringChars(env, jstr, &isCopy);
…
if (isCopy) {
  (*env)->ReleaseStringChars(env, jstr, cstr);
}
```

Figure 4: An improper example retaining a VM resource

2.3 Using Invalid Local References

There is a design pattern in which one native method stores a Java object into a global variable at static initialization time and another method uses it later. Because the reference to the Java object lives across method calls, it must be a global reference rather than a local reference. However, some programmers forget to convert the local reference to the class object into a global one as Figure 5. The local reference should be converted into a global reference by the `NewGlobalRef` function before assigned.

```
static jclass fooCls;
JNIEXPORT void JNICALL
Java_Foo_initialize(JNIEnv *env, jclass cls)
{ /* fooCls should be converted into global */
  fooCls = (*env)->FindClass(env, "Foo");
}
JNIEXPORT void JNICALL
Java_Foo_bar(JNIEnv *env, jobject object)
{
/*fooCls is no longer valid */
jmethodID mid =
(*env)->GetMethodID(env, fooCls, "foo", "()V");
  …
}
```

Figure 5: An example using an invalid local reference

2.4 Calling a JNI Function in a Critical Region

The fourth thing programmers need to care is to avoid JNI function calls in a critical region which starts with a call to `GetStringCritical/GetPrimitiveArrayCritical` and ends with `ReleaseStringCritical` / `ReleasePrimitiveArrayCritical`. Such a JNI function call between these calls to critical functions, for example, as shown in Figure 6, may cause the current thread to block.

Note that we could overlap (not necessarily nest) multiple pairs of `GetStringCritical` (`GetPrimitiveArrayCritical`) and `ReleaseStringCritical` (`ReleasePrimitiveArrayCritical`). This situation could make programmers write more invalid function calls than usual.

Although this mistake is not mentioned in the Traps and Pitfalls chapter of [10][1], we think it is as important as the other mistakes.

```
CallInCriticalRegion(JNIEnv *env,
jobject obj, jstring jstr)
{
  jboolean isCopy;
  const jchar *cstr =
     env->GetStringCritical(jstr, &isCopy);
  /* Any JNI function must not called here */
  env->CallVoidMethod(obj, mid);
  env->ReleaseStringCritical(jstr, cstr);
}
```

Figure 6: An improper example calling a JNI function in a critiacal region

3. DETECTING JNI MISTAKES

Below, we describe the design of our analysis for each JNI problem. Our approach is based on typestate analysis [13] and syntax checking.

3.1 Typestate Analysis for Exception

First, we present an analysis which detects lacks of exception checks between Java method invocations. This analysis is based on typestate analysis and its typestate configuration is defined as Figure 7.

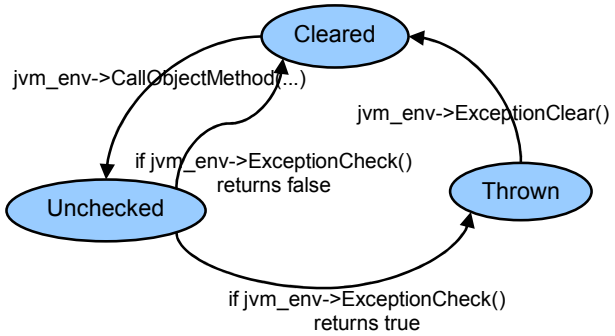

Figure 7: Typestate configuration for the exception analysis

In this analysis, three states are defined: Cleared in which no exception is pending, Unchecked in which it is unknown whether an exception is thrown or not, and Thrown in which an exception is pending. Calls to JNI functions which invoke Java methods (e.g `CallObjectMethod`) trigger transition to the Unchecked state. Also, these function calls are required to be in the Cleared state. If it is in the Unchecked state, the analysis reports an error because an exception might be pending before the call and ignored during the call. Of course, if it is in the Thrown state, the analysis also reports the same error. The state transits from Unchecked to Cleared if an `ExceptionCheck` call returns false. Otherwise it goes to Thrown. In order to go back to the Cleared state from the Thrown state, a call to

[1] We suspect that the reason this problem is not adopted is it is relatively a new problem. These critical JNI function are newly introduced in JDK 1.2

`ExceptionClear` is required. If native code finishes at the Unchecked or Thrown state, we should not report any error because the pending exception will be handled by Java code correctly.

This typestate analysis can be solved as a dataflow problem. Its lattice would be the powerset lattice of $2^{\{Cleared,\ Unchecked,\ Thrown\}}$ ordered by inclusion. The transfer function changes the states as described above.

3.2 Typestate Analysis for Virtual Machine Resources

Errors related virtual machine resources can be detected by another simple typestate analysis configured as Figure 8. A memory block returned by an allocator function immediately enters into the Allocated state. It goes to the Deallocated state when a corresponding deallocator function is called on that memory block.

If a memory block remains in the Allocated state at the end of native code, the analysis reports a memory leak. In addition, it also reports an error if a Deallocated memory block is accessed or passed to a deallocator function.

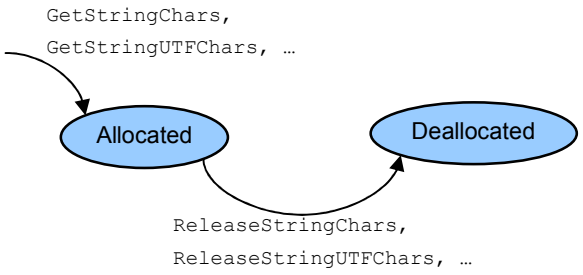

Figure 8: Typestate configuration for VM resources

Although the above configuration is efficient to detect leaks, it produces a false positive for code shown in Figure 9. This is because it does not take into account allocation failure. The allocation functions do not allocate the resource if their invocation has thrown an exception. Although the exception, which is of the `OutOfMemoryError` type, is seldom raised, we can avoid generating a false positive by using the typestate configuration depicted in Figure 10.

```
utf = env->GetStringUTFChars(str, NULL);
if (env->ExceptionOccurred())
  return NULL;
```

Figure 9: An example from which the simple configuration produces a false positive

In this configuration, if invocation of the allocation functions returns NULL or throws an exception, the first argument for the JNI environment goes into the Thrown state. Otherwise it goes into the Cleared state and the allocated resource moves equally to Figure 8. Note that the Thrown and Cleared states are identical to the ones shown in Figure 7 and also used by the exception analysis described before.

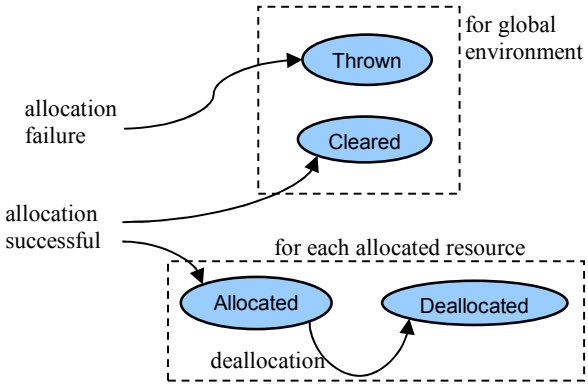

Figure 10: Typestate configuration for the exception of VM resource allocation. The upper states are used for the VM environment (i.e. the first or 'this' argument of the allocator functions) and the lower states are used for each resource allocated successfully (i.e. the return value of the allocator functions).

3.3 Syntax Check for Using Invalid Local References

The main cause of using an invalid local reference is its assignment into a global variable. We detect such an assignment by a syntax checker. It just finds an assignment whose left hand side is a global variable and right hand side takes the value returned by a function call other than `NewGlobalRef` and `NewWeakGlobalRef`.

This simple analysis generates a false positive for code shown in Figure 11 where the global variable is overwritten by a global reference. To avoid generating this false positive and to completely find true positives, we should have reaching definition analysis. However, we selected the syntax checker because its implementation cost is significantly smaller than dataflow analysis. We simply ignore a global variable which has multiple assignments in a function.

```
static jclass fooCls;
JNIEXPORT void JNICALL
Java_Foo_initialize(JNIEnv *env, jclass cls)
{
  fooCls = (*env)->FindClass(env, "Foo");
  fooCls = (*env)->NewGlobalRef(env, fooCls);
}
```

Figure 11: An example correctly assigning a global reference

3.4 Typestate Analysis for Calling a JNI Function in a Critical Region

A function call in a critical region can be detected by yet another typestate analysis that is configured as Figure 12. There are two states, Critical and NotCritical. When the `GetStringCritical` or `GetPrimitiveArrayCritical` function is called, the native code unconditionally goes into the

Critical state regardless of the current state. When the `ReleaseStringCritical` or `ReleasePrimitiveArrayCritical` function is called, it unconditionally goes to the NotCritical state regardless of the current state. Other JNI function calls are required to be in the NotCritical state. This problem can be also solved as a dataflow problem by converting this typestate configuration into a powerset lattice.

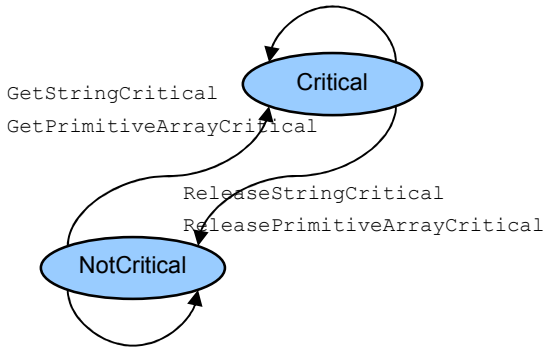

Figure 12: Typestate configuration for Calling a JNI Function in a Critical Region

This typestate analysis is sound but not complete because it cannot detect an absence of the release functions when critical regions are overlapped. In order to deal with overlapping, more accurate typestate configuration is necessary. Figure 13 shows typestate configuration for at most three critical regions. In this configuration 1, 2 and 3+ which represent the number of overlaps are introduced instead of Critical. Transitions are conditional on the current state and the native method function is required to be in the NotCritical state at the beginning. This configuration is more accurate than the previous one but still incomplete.

The complete configuration should have infinite states, NotCritical, 1, 2, 3, … like a counter automaton with a variable for the number of overlaps. When implementing this typestate analysis with this infinite configuration on a dataflow analysis framework, a powerset lattice cannot be used because of infinite height. Instead, a flat lattice is used. We will discuss if this completeness matters in real applications in Section 5.

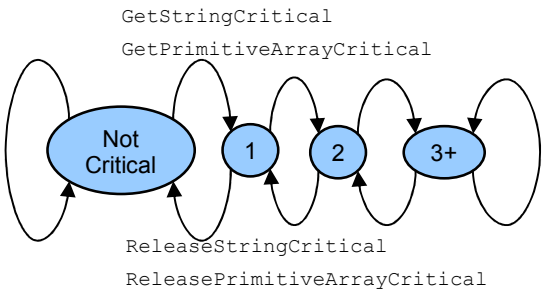

Figure 13: Typestate configuration for at most three overlapping critical regions

4. IMPLEMENTATION

We have implemented the typestate analysis described in Section 3.1, and 3.4 on BEAM. BEAM has extensibility for typestate checking by a user-defined typestate configuration. In this configuration, users can specify requirements for a function, namely states in which an argument of a function must be before called. They can also specify the effect of a function, namely states an argument of the function goes after called. Although Figure 7 just describes about `CallVoidMethod`, `ExceptionCheck`, and `ExceptionClear`, we have defined typestate configuration for all the JNI functions which are defined as members of a struct in C++.

BEAM has built-in typestate configuration similar to Figure 8 and allows users to write specification about memory allocator and deallocator functions. For the typestate analysis described in Section 3.2, we have defined such attributes for C++ JNI functions listed in section 2.2. With those attributes, BEAM can detect virtual machine memory leaks, multiple deallocations of a resource and use of deallocated resources.

C++:

```
mid = env->GetMethodID(cls, "foo", "()V");
env->CallVoidMethod(obj, mid)
```

C:

```
mid =(*env)->GetMethodID(env, cls, "foo",
                                    "()V");
(*env)->CallVoidMethod(env, obj, mid)
```

Figure 14: Function call difference between C++ and C

In C, JNI functions are called indirectly through a pointer in a struct as shown in Figure 14. Ideally, we should resolve which function can be called at an indirect call site by pointer analysis. However, we did not find any assignment to the pointer member variables except for the initialization of the `env` struct in any application or JVM code. Therefore, we have extended BEAM as it can interpret typestate configuration defined for pointer variables and gave the same properties as ones defined for C++ functions.

BEAM has capability of interprocedural analysis (IPA). In the first phase, BEAM's IPA analyzes a function with our typestate configuration and generates some information about requirements for calling context and a set of possible states after a call. The second phase analyzes a function, which may call another function, with the information obtained by the previous phase. Unfortunately, the analysis in the first phase is flow-insensitive. Therefore, it generates less accurate information than flow-sensitive one. For example, the flow-insensitive analysis is not able to find out the function `CallFooWrapper2` in Figure 15 finishes in the Unchecked state whereas the flow-sensitive analysis is.

In addition to typestate analysis, we have implemented the syntax checker described in Section 3.3 on BEAM as well. BEAM allows developers to implement their own checkers analyzing its internal representations, namely its abstract syntax tree and flowgraph. We decided to implement our syntax checker to analyze the flowgraph. BEAM represents a JNI function call in C in flowgraph in the same way as the equivalent call in C++, while this is not the case in the abstract syntax tree. For example, C and C++ JNI function calls in Figure 14 are converted into a flowgraph node called *CALL* with the same three incoming edges.

Thus, we need to have only one implementation for both languages. Even if native code is written in a third language, we will be able to reuse the implementation. Another reason we chose the flowgraph as our analysis target is that BEAM does optimization on the flowgraph and it brings us efficiency.

```
void CallFooWrapper2(JNIEnv *env, jobject obj)
{
  jclass cls = env->GetObjectClass(obj);
  jmethodID mid =
     env->GetMethodID(cls, "foo", "()V");
  env->CallObjectMethod(obj, mid);
  if (env->ExceptionCheck()) {
    …
    env->ExceptionClear();
  }
  env->CallObjectMethod(obj, mid);
}
```

Figure 15: An example which might throw an exception

5. EXPERIMENTAL RESULTS

We executed our JNI analysis tool on four open source software projects shown in Table 2. We analyzed source code snapshots obtained from their code repository. Their SVN/ BZR revisions or CVS dates are shown in Column 2. In this experiment, we only analyzed files which include a JNI environment access (i.e. ones in which "JNIEnv" is found by grep.) The numbers of such files and their lines are shown in Columns 4 and 5, respectively. We first try the simple configurations shown in Figure 8 and Figure 12 for the detection of VM resource leaks and calling JNI function call in a critical region, respectively.

Table 2: Benchmark programs

	revision/ CVS date	description	# of files	# of lines
Harmony [1]	r566512	Java class library	159	53159
Java Gnome [9]	411	Java interface to Gnome	20	3670
Gnu Classpath [7]	2008-01-22	Java class library	69	27364
Mozilla Firefox [5]	2008-01-22	web browser	22	22427

In addition to our JNI experiments, we executed BEAM without our JNI analysis features on the same file set to see how much the JNI analysis features cost. It ran at the same speed and generated the same non-JNI bug reports. Thus we do not report experimental results of BEAM by itself. We can conclude that our JNI analysis feature does not make a penalty of both precision and speed.

Table 3 shows results of our analyses on the benchmark programs. We ran BEAM against these programs under Linux 2.6 on an unloaded IBM IntelliStation M Pro which contains a 2.66 GHz Intel Core2 processor and 3GB of RAM. Row 2 represents the number of bugs which are not related to our JNI analysis. They include array index out of range, null dereference, and use of an uninitialized variable. Rows 3 to 6 correspond to mistakes described in Section 2.1 through 2.4. We manually verified all the reported errors and found in Gnu Classpath only one false positive of a VM resource leak caused by the code like Figure 9. However, if we use the more complex configuration shown in Figure 10, the false positive disappeared. We do not report its experimental results because they did not differ from the run with the simpler configuration except for the false positive, and the analysis time was almost the same.

Table 3: Experimental results (numbers in parentheses were found by interprocedural analysis)

	Harmony	Java Gnome	Gnu Classpath	Mozilla Firefox	Total	
Non JNI	84	3	12	15	114	
Error Checking	22	2	22(10)	9(3)	55	86
Virtual Machine Resource	18	0	7	0	25	
Using Invalid Local References	4	0	0	0	4	
JNI Function Call In a Critical Region	2 (1)	N/A	N/A	N/A	2	
Time (mm:ss)	31:59	0:12	1:59	2:02		

At a first glance, the table shows that our JNI analysis totally generates 43% of all the bug reports. In other words, by adding our JNI-domain-specific analysis to a general bug-finding tool the number of reported bugs increased by 76% for JNI-domain-specific programs.

Looking at the breakdown, the kind of errors most frequently reported is error checking shown in Row 3. This coincides with what Section 10.1 of [10] points out. These benchmarks have many Java method invocation sites, most of which are followed by the error checking code. However, the programmers still forgot to insert such error checking code in some places which our tool attempts to detect.

The second most frequently reported errors are virtual machine resource problems. As mentioned before, most of GNU Classpath's memory leaks were caused by programmers' misunderstanding of the allocator functions.

We found four assignments of local references into a global variable in Harmony. We verified that the global variable is used like a local variable and these local references are correctly used in terms of results. In this regard, these four reports are false positives. However, for better coding we think that these variables should be declared local.

We found two JNI function calls in critical regions in Harmony. As mentioned before, critical regions can be overlapped. Harmony includes 15 critical regions which overlap with each other. We tried the typestate analysis with the infinite state configuration for Harmony. However, BEAM reported the same set of JNI errors and almost the same set of non-JNI ones, while it took 41 minutes and 7 seconds. We think that this increased analysis time is because we had to give to each native method function a precondition that requires the function to start with the NotCritical state. This precondition could become extra information and make BEAM take more time.

Finally, we show the effectiveness of IPA. Our IPA relies on BEAM's IPA, which is flow-insensitive. First we were not sure how effective the IPA was because we predicted there were many wrapper procedures like Figure 15. However, the flow-insensitive IPA is working somewhat effectively for Gnu Classpath and Mozilla. The reason is in those applications callee functions are very simple from a viewpoint of JNI. Most of them call a JNI method invocation function just once. Otherwise, their control flows related to JNI are simple. We still need to investigate why IPA was not so effective for Harmony and Java Gnome; they might have few simple wrapper functions or require flow-sensitive IPA. However, looking at the running time, we believe that our flow-insensitive IPA yields us good balanced results.

6. RELATED WORK

Prior work dealing with JNI can be categorized into runtime checks, static analysis for safety, and new programming models for integrating two programming language.

6.1 Runtime Checking by Java VM

While this paper is focused on static analysis, runtime checking by the –Xcheck:jni option of Java VMs is also helpful for programmers. We ran a few test programs with two commercial VMs, Sun HotSpot(TM) Client VM [14] version 1.5.0 and IBM J9 VM [2] version 1.5.0.

The test programs are as follows:

- In the "Error Checking" tests, we ran the native code shown in Figure 2 with two cases. In the "Throw" case, we made the first Java method invocation (call `foo()`) actually throw an exception. In the "Not Throw" case, we did not.
- In the "VM Resource – Leak" test, we called only the `GetStringChars` function with the `isCopy` parameter NULL and never called the `ReleaseStringChars` function.
- In the "VM Resource – Release Twice" test, we called `GetStringChars` and called `ReleaseStringChars` twice.
- In the "VM Resource – Access Released Resource" test, we called `GetStringChars` followed by `ReleaseStringChars`, and read the contents of the array already released.
- In the "Using Invalid Local Reference" test, we ran the native code shown in Figure 5.
- In the "Call in Critical Region" test, we ran the native code shown in Figure 6.

Table 4 shows the results of these tests on the two VMs. The finding here is that the –Xchceck:jni option, especially of J9, is useful for the mistakes we try to detect. However, in order to find bugs by using VM's runtime checks, failure-exposing test inputs must be provided whereas our techniques require only source code.

Table 4: Runtime checking by Java VM

		HotSpot VM		IBM J9 VM	
		opt	noopt	with opt	noopt
Error Checking	Throw	error	crash	error	crash
	Not Throw			warning	
VM Resource	Leak			warning	
	Release Twice	crash	crash	error	crash
	Access Released Resource	read0	read0	crash	crash
Using Invalid Local Reference		error	crash	error	crash
Call in Critical Region		warning		error	

opt: run with the –Xcheck:jni option. *noopt*: run withouth the option. *error*: exit with an error message. *warning*: continue running with a warning message. *crash*: aborted with an fatal error such as segmentation fault. *blank cell*: continue running silently. *read0*: the released resource is filled with zero and further access can read it.

6.2 Static Analysis Tools for JNI

Furr and Foster [6] have presented a polymorphic type inference system for JNI, which statically analyzes native code and checks the correctness of literal names given to JNI function calls. We implemented their type system, which we found very useful at build time. It can detect misspelling before testing. In our experiments, however, it did not find any serious error but a minor one which misspells `"java/net/ServerSocket"` as `"Ljava/net/ServerSocket;"` in a call to the `FindClass` function. That might be because the code checked in a repository is well-tested. Tan et al.[15] proposed a framework called Safe Java Native Interface that ensures validity of Java references. For example, it makes sure the Java references are not destroyed by pointer operations. Both of the analyses are beneficial to ensure correctness of JNI function invocations. On the other hand, our analysis deals with a broader set of mistakes including exception handling mistakes, memory leaks, using invalid local references and JNI function calls in a critical region.

Livshits et al.[11] proposed a technique to resolve invocation targets from Java reflection calls. They resolve targets using points-to information and available type declarations in Java. Although they do not claim, we believe their work can be applied to static analysis for JNI.

6.3 New Programming Models and Runtime

There are efforts [4][8] introducing new programming languages as mixture of Java and C to bring programmers productivity and safety. In their languages both Java code and C code can be nested in each other and programmers do not need to write native code with JNI function calls. The correctness of exception handling as well as syntax is checked by the compilers. As long as the compiler generates correct code, there is no memory leak or invalid use of local references or JNI function calls in a critical region. However, since they are quite new programming environments, it requires programmers' migration efforts.

On the other hand, Jace [16] helps simplify JNI programming by providing C++ proxy classes for Java classes. As long as native code accesses Java objects through the proxy classes, all the type errors can be checked by a C++ compiler as naming errors. In addition, because the proxy classes convert a Java exception into a C++ proxy class and throw it, exception correspondence can also be checked by a C++ compiler. However, Jace also requires migration efforts. Besides, new proxy classes must be provided for safe accesses to new Java classes.

Whereas Jace handles Java exceptions at the class library level, CEE-J [12] supports mixing C++ and Java exceptions at the virtual machine level. This means that programmers writing native methods in C++ can throw Java exceptions with the C++ 'throw' statements. Similarly, C++ code can catch Java exceptions with the C++ 'try'/'catch' mechanism. Furthermore, Java code can catch an exception thrown from C++. This shared exception support can enhance the readability of native code. However, this mechanism works only on the CEE-J virtual machine.

7. CONCLUSION

We presented static analysis techniques for JNI programming mistakes which are not dealt with by existing tools. We implemented a JNI bug-finding tool with those techniques, and showed that our tool could find many errors in real applications using JNI.

8. ACKNOWLEDGMENTS

We thank members of the BEAM team, Dan Brand, Florian Krohm, Frank Wallingford and John Darringer for comments on the paper.

9. REFERENCES

[1] The Apache Software Foundation, Apache Harmony - Opensource Java SE, http://harmony.apache.org/

[2] Bailey, C. Java technology, IBM style: Introduction to the IBM developer kit, http://www-128.ibm.com/developerworks/java/library/j-ibmjava1.html, May, 2006.

[3] Brand, D. A Software Falsifier, In Proc. the 11th International Symposium on Software Reliability Engineering (ISSRE'00), 2000

[4] Bubak, M., Kurzyniec, D., Luszczek, P., Sunderam, V. Creating Java to Native Code Interfaces with Janet, In Scientific Programming, Volume 9 , Issue 1, Jan., 2001.

[5] Firefox web browser, http://www.mozilla.org/firefox/

[6] Furr, M., Foster, J. Polymorphic Type Inference for the JNI, In Proc. 16th European Symposium on Programming (ESOP'06), Vienna, Austria. Mar., 2006.

[7] Gnu Classpath, http://www.gnu.org/software/classpath/

[8] Hirzel, M., Grimm, R. Jeannie: Granting Java Native Interface Developers Their Wishes, In Proc. the 22nd annual ACM SIGPLAN conference on Object Oriented Programming Systems and Applications (OOPSLA 2007), Oct., 2007.

[9] Opening GTK and GNOME to Java Programmers, http://java-gnome.sourceforge.net/

[10] Liang, S. The Java Native Interface: Programmer's Guide and Specification, Addison-Wesley, Reading, MA, 1997.

[11] Livshits, B., Whaley, J., Lam, M., S. Reflection Analysis for Java, In Proc. the Third Asian Symposium on Programming Languages and Systems, Nov., 2005.

[12] Skelmir, LLC. SKELMIR virtual machine technology, http://www.skelmir.com/products/ceej.html

[13] Strom, R., E., Yemini, S. Typestate: A programming language concept for enhancing software reliability, IEEE Transactions on Software Engineering, vol. 12, issue 1, Jan., 1986.

[14] Sun Microsystems, Inc. Java SE HotSpot at a glance, http://java.sun.com/javase/technologies/hotspot/

[15] Tan, G., Appel, A. W., Chakradhar, S., Raghunathan, A., Ravi, S., Wang, D. Safe Java native interface. In Proc. 2006 IEEE International Symposium on Secure Software Engineering, Mar., 2006.

[16] Tzabari, G. Jace - JNI Made Easy, http://reyelts.dyndns.org:8080/jace/release/docs/index.html